Dependable Computing

John Rushby Editor

Evaluation of Safety-Critical Software

Methods and approaches for testing the reliability and trustworthiness of software remain among the most controversial issues facing this age of high technology. The authors present some of the crucial questions faced by software programmers and eventual users.

David L. Parnas, A. John van Schouwen, and Shu Po Kwan

It is increasingly common to use programmable computers in applications where their failure could be life-threatening and could result in extensive damage. For example, computers now have safety-critical functions in both military and civilian aircraft, in nuclear plants, and in medical devices. It is incumbent upon those responsible for programming, purchasing, installing, and licensing these systems to determine whether or not the software is ready to be used. This article addresses questions that are simple to pose but hard to answer. What standards must a software product satisfy if it is to be used in safety-critical applications such as those mentioned? What documentation should be required? How much testing is needed? How should the software be structured?

This article differs from others concerned with software in safety-critical applications, in that it does not attempt to identify *safety* as a property separate from reliability and trustworthiness. In other words, we do not attempt to separate safety-critical code from other code in a product used in a safety-critical application. In our experience, software exhibits *weak-link* behavior, that is failures in even the unimportant parts of the code can have unexpected repercussions elsewhere. For a discussion of another viewpoint, we suggest the work of N. G. Leveson [6, 7, 8].

We favor keeping safety-critical software as small and simple as possible by moving any functions that are not safety critical to other computers. This further justifies our assumption that all parts of a safety-critical software product must be considered safety critical.

WHY IS SOFTWARE A SPECIAL CONCERN?

Within the engineering community software systems have a reputation for being undependable, especially in the first years of their use. The public is aware of a few spectacular stories such as the Space Shuttle flight that was delayed by a software timing problem, or the Venus probe that was lost because of a punctuation error. In the software community, the problem is known to be much more widespread.

A few years ago, David Benson, professor of Computer Science at Washington State University, issued a challenge by way of several electronic bulletin board systems. He asked for an example of a real-time system that functioned adequately when used for the first time by people other than its developers for a purpose other than testing. Only one candidate for this honor was proposed, but even that candidate was controversial. It consisted of approximately 18,000 instructions, most of which had been used for several years before the "first use." The only code that had not been used before that first use was a simple sequence of 200 instructions that simulated a simple analogue servomechanism. That instruction sequence had been tested extensively against an analogue model. All who have looked at this program regard it as exceptional. If we choose to regard this small program as one that worked in its first real application, it is the proverbial "exception that proves the rule."

As a rule software systems do not work well until they have been used, and have failed repeatedly, in real applications. Generally, many uses and many failures are required before a product is considered reliable. Software products, including those that have become relatively reliable, behave like other products of evolution-like processes; they often fail, even years after they were built, when the operating conditions change.

While there are errors in many engineering products, experience has shown that errors are more common, more pervasive, and more troublesome, in software than in other technologies. This information must be understood in light of the fact it is now standard practice among software professionals to have their product go through an extensive series of carefully planned tests before real use. The products fail in their first real use because the situations that were not anticipated by the programmers were also overlooked by the test planners. Most major computer-using organizations, both

This work was supported by the National Science and Engineering Research Board of Canada as well as the Atomic Energy Control Board of Canada.

military and civilian, are investing heavily in searching for ways to improve the state of the art in software. The problem remains serious and there is no sign of a "silver bullet." The most promising development is the work of Harlan Mills and his colleagues at IBM on a software development process known as "clean room" [3, 9, 12]. Mills uses randomly selected tests, carried out by an independent testing group. The use of randomly generated test data reduces the likelihood of shared oversights. We will discuss this approach in more detail later in this article.

WHY IS SOFTWARE USED?

If software is so untrustworthy, one might ask why engineers do not avoid it by continuing to use hard-wired digital and analogue hardware. Here, we list the three main advantages of replacing hardware with software:

1. Software technology makes it practical to build more *logic* into the system. Software-controlled computer systems can distinguish a large number of situations and provide output appropriate to each of them. Hard-wired systems could not obtain such behavior without prohibitive amounts of hardware. Programmable hardware is less expensive than the equivalent hard-wired logic because it is regular in structure and it is mass produced. The economic aspects of the situation also allow software-controlled systems to perform more checking; reliability can be increased by periodic execution of programs that check the hardware.
2. Logic implemented in software is, in theory, easier to change than logic implemented in hardware. Many changes can be made without adding new components. When a system is replicated or located in a physical position that is hard to reach, it is far easier to make changes in software than in hardware.
3. Computer technology and software flexibility make it possible to provide more information to operators and to provide that information in a more useful form. The operator of a modern software-controlled system can be provided with information that would be unthinkable in a pure hardware system. All of this can be achieved using less space and power than was used by noncomputerized systems.

These factors explain the replacement of hard-wired systems with software-controlled systems in spite of software's reputation as an unreliable technology.

HOW ARE SOFTWARE CONTROLLERS LIKE OTHER CONTROLLERS?

In the next section we will argue that software technology requires some refinements in policies and standards because of differences between software and hardware technology. However, it is important to recognize some common properties of software and hardware control systems.

In the design and specification of control systems, engineers have long known how to use a black box mathematical model of the controller. In such models, (1) the inputs to the controller are described as mathematical functions of certain observable environmental state variables, (2) the outputs of the controller are described as mathematical functions of the inputs, (3) the values of the controlled environmental variables are described as mathematical functions of the controller's outputs, and (4) the required relation between the controlled variables and observed variables is described. It is then possible to confirm that the behavior of the controller meets its requirements.

It is important to recognize that, in theory, software-implemented controllers can be described in exactly the same way as black box mathematical models. They can also be viewed as black boxes whose output is a mathematical function of the input. In practice, they are not viewed this way. One reason for the distinction is that their functions are more complex (i.e. harder to describe) than the functions that describe the behavior of conventional controllers. However, [4] and [17] provide ample evidence that requirements for real systems can be documented in this way. We return to this theme later.

HOW IS SOFTWARE DIFFERENT FROM OTHER CONTROLLER TECHNOLOGIES?

Software problems are often considered growing pains and ascribed to the adolescent nature of the field. Unfortunately there are fundamental differences between software and other approaches that suggest these problems are here to stay.

Complexity: The most immediately obvious difference between software and hardware technologies is their complexity. This can be observed by considering the size of the most compact descriptions of the software. Precise documentation, in a reasonably general notation, for small software systems can fill a bookcase. Another measure of complexity is the time it takes for a programmer to become closely familiar with a system. Even with small software systems, it is common to find that a programmer requires a year of working with the program before he/she can be trusted to make improvements on his/her own.

Error Sensitivity: Another notable property of software is its sensitivity to small errors. In conventional engineering, every design and manufacturing dimension can be characterized by a tolerance. One is not required to get things exactly right; being within the specified *tolerance* of the right value is good enough. The use of a tolerance is justified by the assumption that small errors have small consequences. It is well known that in software, trivial clerical errors can have major consequences. No useful interpretation of tolerance is known for software. A single punctuation error can be disastrous, even though fundamental oversights sometimes have negligible effects.

Hard to Test: Software is notoriously difficult to test

adequately. It is common to find a piece of software that has been subjected to a thorough and disciplined testing regime has serious flaws. Testing of analogue devices is based on interpolation. One assumes that devices that function well at two close points will function well at points in-between. In software that assumption is not valid. The number of cases that must be tested in order to engender confidence in a piece of software is usually extremely large. Moreover, as Harlan Mills has pointed out, "testing carried out by selected test cases, no matter how carefully and well-planned, can provide nothing but anecdotes" [3, 9, 12].

These properties are fundamental consequences of the fact that the mathematical functions implemented by software are not continuous functions, but functions with an arbitrary number of discontinuities. The lack of continuity constraints on the functions describing program effects makes it difficult to find compact descriptions of the software. The lack of such constraints gives software its flexibility, but it also allows the complexity. Similarly, the sensitivity to small errors, and the testing difficulties, can be traced to fundamental mathematical properties; we are unlikely to discover a miracle cure. Great discipline and careful scrutiny will always be required for safety-critical software systems.

Correlated Failures: Many of the assumptions normally made in the design of high-reliability hardware are invalid for software. Designers of high-reliability hardware are concerned with manufacturing failures and wear-out phenomena. They can perform their analysis on the assumption that failures are not strongly correlated and simultaneous failures are unlikely. Those who evaluate the reliability of hardware systems should be, and often are, concerned about design errors and correlated failures; however in many situations the effects of other types of errors are dominant.

In software there are few errors introduced in the manufacturing (compiling) phase; when there are such errors they are systematic, not random. Software does not wear out. The errors with which software reliability experts must be concerned are design errors. These errors cannot be considered statistically independent. There is ample evidence that, even when programs for a given task are written by people who do not know of each other, they have closely related errors [6, 7, 8].

In contrast to the situation with hardware systems, one cannot obtain higher reliability by duplication of software components. One simply duplicates the errors. Even when programs are written independently, the oversights made by one programmer are often shared by others. As a result, one cannot count on increasing the reliability of software systems simply by having three computers where one would be sufficient [6, 7, 8].

Lack of Professional Standards: A severe problem in the software field is that, strictly speaking, there are no software engineers. In contrast to older engineering fields, there is no accrediting agency for professional software engineers. Those in software engineering have not agreed on a set of skills and knowledge that should be possessed by every software engineer. Anyone with a modicum of programming knowledge can be called a software engineer. Often, critical programming systems are built by people with no postsecondary training about software. Although they may have useful knowledge of the field in which the software will be applied, such knowledge is not a substitute for understanding the foundations of software technology.

SOFTWARE TESTING CONCERNS

Some engineers believe one can design black box tests without knowledge of what is inside the box. This is, unfortunately, not completely true. If we know that the contents of a black box exhibit linear behavior, the number of tests needed to make sure it would function as specified could be quite small. If we know that the function can be described by a polynomial of order "*N*," we can use that information to determine how many tests are needed. If the function can have a large number of discontinuities, far more tests are needed. That is why a shift from analogue technology to software brings with it a need for much more testing.

Built-in test circuitry is often included in hardware to perform testing while the product is in use. Predetermined values are substituted for inputs, and the outputs are compared to normative values. Sometimes this approach is imitated in software designs and the claim is made that built-in online testing can substitute for black box testing. In hardware, built-in testing tests for decay or damage. Software does not decay and physical damage is not our concern. Software can be used to test the hardware, but its value for testing itself is quite doubtful. Software self-testing does increase the complexity of the product and, consequently, the likelihood of error. Moreover, such testing does not constitute adequate testing because it usually does not resemble the conditions of actual use.

The fundamental limitations on testing mentioned earlier have some very practical implications.

We cannot test software for correctness: Because of the large number of states (and the lack of regularity in its structure), the number of states that would have to be tested to assure that software is correct is preposterous. Testing can show the presence of bugs, but, except for toy problems, it is not practical to use testing to show that software is free of design errors.

It is difficult to make accurate predictions of software reliability and availability: Mathematical models show that it is practical to predict the reliability of software, provided that one has good statistical models of the actual operating conditions. Unfortunately, one usually gains that information only after the system is installed. Even when a new system replaces an existing one, differences in features may cause changes in the input distribution. Nonetheless, in safety-critical situations, one must attempt to get and use the necessary statistical

data. The use of this data is discussed later in this article.

Predictions of availability are even more difficult; estimates of availability depend on predictions of the time it will take to correct a bug in the software. We never know what that amount of time will be in advance; data from earlier bugs is not a good predictor of the time it will take to find the next bug.

It is not practical to measure the trustworthiness of software: We consider a product to be trustworthy if we believe that the probability of it having a potentially catastrophic flaw is acceptably low. Whereas reliability is a measure of the probability of a problem occurring while the system is in service, trustworthiness is a measure of the probability of a serious flaw remaining after testing and review. In fact, inspection and testing can increase the trustworthiness of a product without affecting its reliability.

Software does not need to be correct in order to be trustworthy. We will trust imperfect software if we believe its probability of having a serious flaw is very low. Unfortunately, as we will show, the amount of testing necessary to establish high confidence levels for most software products is impractically large. The number of states and possible input sequences is so large that the probability of an error having escaped our attention will remain high even after years of testing. Methods other than testing must be used to increase our trust in software.

There is a role for testing: A number of computer scientists, aware of the limitations on software testing, would argue that one should not test software. They would argue that the effort normally put into testing should, instead, be put into a form of review known as mathematical verification. A program is a mathematical object and can be proven correct. Unfortunately, such mathematical inspections are based on mathematical models that may not be accurate. No amount of mathematical analysis will reveal discrepancies between the model being used and the real situation; only testing can do that. Moreover, errors are often made in proofs. In mature engineering fields, mathematical methods and testing are viewed as complementary and mutually supportive.

There is a need for an independent validation agency: It is impossible to test software completely and difficult to test one's own design in an unbiased way. A growing number of software development projects involve independent verification and validation (V&V). The V&V contractor is entirely independent of the development contractor. Sometimes a competitor of the development contractor is given the V&V contract. The testers work from the specification for the software and attempt to develop tests that will show the software to be faulty. One particularly interesting variation of this approach has been used within the IBM Federal Systems Division. In IBM's *clean room* development approach the authors of the software are not allowed to execute their programs. All testing is done by an independent tester and test reports are sent to the developer's supervisors. The test cases are chosen using random number generators and are intended to yield statistically valid data. It was hypothesized that the software would be written far more carefully under these conditions and would be more reliable. Early reports support the hypothesis [3, 9, 12].

It is important that these validation tests not be made available to the developers before the software is submitted for testing. If the developers know what tests will be performed, they will use those tests in their debugging. The result is likely to be a program that will pass the tests but is not reliable in actual use.

SOFTWARE REVIEWABILITY CONCERNS

Why is reviewability a particular concern for software?

Traditionally, engineers have approached software as if it were an art form. Each programmer has been allowed to have his own style. Criticisms of software structure, clarity, and documentation were dismissed as "matters of taste."

In the past, engineers were rarely asked to examine a software product and certify that it would be trustworthy. Even in systems that were required to be trustworthy and reliable, software was often regarded as an unimportant component, not requiring special examination.

In recent years, however, manufacturers of a wide variety of equipment have been substituting computers controlled by software for a wide variety of more conventional products. We can no longer treat software as if it were trivial and unimportant.

In the older areas of engineering, safety-critical components are inspected and reviewed to assure the design is consistent with the safety requirements. To make this review possible, the designers are required to conform to industry standards for the documentation, and even the structure, of the product. The documentation must be sufficiently clear and well organized that a reviewer can determine whether or not the design meets safety standards. The design itself must allow components to be inspected so the reviewer can verify they are consistent with the documentation. In construction, inspections take place during the process—while it is still possible to inspect and correct work that will later be hidden.

When software is a safety-critical component, analogous standards should be applied. In software, there is no problem of physical visibility but there is a problem of clarity. Both practical experience and planned experiments have shown that it is common for programs with major flaws to be accepted by reviewers. In one particularly shocking experiment, small programs were deliberately flawed and given to a skilled reviewer team. The reviewers were unable to find the flaws in spite of the fact they were certain such flaws were present. In theory, nothing is invisible in a program—

it is all in the listing; in practice, poorly structured programs hide a plethora of problems.

In safety-critical applications we must reject the "software-as-art-form" approach. Programs and documentation must conform to standards that allow reviewers to feel confident they understand the software and can predict how it will function in situations where safety depends on it. However, we must, equally strongly, reject standards that require a mountain of paper that nobody can read. The standards must insure clear, precise, and concise documentation.

It is symptomatic of the immaturity of the software profession that there are no widely accepted software standards assuring the reviewability essential to licensing of software products that must be seen as trustworthy. The documentation standards name and outline certain documents, but they only vaguely define the contents of those documents. Recent U.S. military procurement regulations include safety requirements; while they require that safety checks be done, they neither describe how to do them nor impose standards that make those checks practicable. Most standards for code documentation are so vague and syntactic in nature that a program can meet those standards in spite of being incomprehensible.

In the next section we derive some basic standards by considering the reviews that are needed and the information required by the reviewers.

What reviews are needed?

Software installed as a safety-critical component in a large system should be subjected to the following reviews:

a. Review for correct intended function. If the software works as the programmers intend, will it meet the actual requirements?
b. Review for maintainable, understandable, well documented structure. Is it easy to find portions of the software relevant to a certain issue? Are the responsibilities of the various modules clearly defined? If all of the modules work as required, will the whole system work as intended? If changes are needed in the future, can those changes be restricted to easily identified portions of the code?
c. Review each module to verify the algorithm and data structure design are consistent with the specified behavior. Is the data structure used in the module appropriate for representing the information maintained by that module? If the programs are correctly coded, will the modules perform as required? Will the algorithms selected perform as required? These reviews must use mathematical methods; one cannot rely on intuitive approaches. We have found a formal review based on functional semantics, [10], to be practical and effective.
d. Review the code for consistency with the algorithm and data structure design. Is the actual source code consistent with the algorithms and data structures described by the designers? Have the assemblers, compilers, and other support tools been used correctly?
e. Review test adequacy. Was the testing sufficient to provide sound confidence in the proper functioning of the software?

The structure of this set of reviews is consistent with modern approaches to software engineering. Because we are unable to comprehend all the critical details about a software product at once, it is necessary to provide documentation that allows programmers and reviewers to focus on one aspect at a time and to zoom in on the relevant details.

Developing and presenting these views in the sequence listed is the analogue of providing inspections during a construction project. Just as construction is inspected before further work obscures what has been done, the early specifications should be reviewed before subsequent coding hides the structure in a sea of detail.

The set of reviews also reflects the fact that reviewers of a software product have a variety of skills. Those who have a deep understanding of the requirements are not usually skilled software designers. It follows that the best people to review the functional behavior of the software are not the ones who should study the software. Similarly, within the software field we have people who are good at algorithm design, but not particularly good finding an architecture for software products. Skilled algorithm designers are not necessarily experts on a particular compiler or machine language. Those intimately familiar with a compiler or assembly language are not always good at organizing large programs. When the software is safety critical, it is important that each of the five reviews be conducted by those best qualified to review that aspect of the work.

Within this framework, all code and documentation supplied must be of a quality that facilitates review and allows the reviewers to be confident of their conclusions. It is the responsibility of the designers to present their software in a way that leaves no doubt about their correctness. It is not the responsibility of the reviewers to guess the designers' intent. Discrepancies between code and documentation must be treated as seriously as errors in the code. If the designers are allowed to be sloppy with their documentation, quality control will be ineffective.

In the following sections of this article, we will describe the documentation that must be provided for each of these reviews. This documentation should not be created merely for review purposes. It should be used throughout the development to record and propagate design decisions. When separate review documents are produced, projects experience all the problems of keeping two sets of books. Because of the complexity of software products, it is unlikely that both records would be consistent. Moreover, the documents described below from the reviewers' viewpoint are invaluable to the designers as well [5, 13, 16].

What documentation is required to review the functional requirements?

The software can be viewed as a control system whose output values respond to changes in the states of variables of interest in its environment. For many real-time systems, the desired outputs approximate piece-wise continuous functions of time and the history of the relevant environmental parameters. For other systems, the outputs are functions of a snapshot of the environmental parameters taken at some point in time. Some systems provide both reports and continuous outputs.

The reviewers at this stage should be engineers and scientists who understand the situation being monitored and the devices to be controlled. They may not be computer specialists and should not be expected to read and understand programs. Because the requirements could, in theory, be fulfilled by a completely hardware design, the description should use the mathematics of control systems, not the jargon and notation of computer programming. The functional requirements can be stated precisely by giving three mathematical relations: (1) The required values of the controlled environmental variables in terms of the values of the relevant observable environmental parameters, (2) the computer inputs in terms of those observable environmental variables, and (3) the values of the controlled environmental variables in terms of the computer outputs.

These requirements can be communicated as a set of tables and formulae describing the mathematical functions to be implemented [4]. We should not describe a sequence of computations anywhere in this document. The use of natural language, which inevitably introduces ambiguity, should be minimized. Documents of this form have been written for reasonably complex systems and are essential when safety-critical functions are to be performed. Our experience has shown that documents written this way can be thoroughly and effectively reviewed by engineers who are not programmers. Some suggestions for organizing the reviews are contained in [19]. A complete example of such a document has been published as a model for other projects [17].

What documentation is required to review the software structure?

For this review we require documents that describe the breakdown of the program into modules. Each module is a unit that should be designed, written and reviewed independently of other modules. Each module is a collection of programs; the programs that can be invoked from other modules are called access programs. The purpose of this review is to make sure that: (1) the structure is one that allows independent development and change; (2) all programs that are needed are included once and only once in the structure; (3) the interfaces to the modules are precisely defined; (4) the modules are compatible and will, together, constitute a system that meets the functional requirements.

For this review three types of documents are required. The first is the requirements specification, which should have been approved by an earlier review. The second is an informal document describing the responsibilities of each module. The purpose of this *module guide* is to allow a reviewer to find all the modules relevant to a particular aspect of system design [1]. The third type of document is known as a module specification. It provides a complete black box description of the module interface. There should be one specification for each module mentioned in the module guide [2, 14].

Reviewers of these documents must be experienced software engineers. Some of them should have had experience with similar systems. This experience is necessary to note omissions in the module structure. Discussions of these documents and how to organize the reviews are contained in [14, 19].

What documentation is required to review the module's internal design?

The first step in designing the module should be to describe the data structures that will be used and each proposed program's effect on the data. This information can be described in a way that is, except for the data types available, independent of the programming language being used.

The design documentation is a description of two types of mathematical functions: program functions and abstraction functions. This terminology was used in IBM's Federal Systems Division, the IBM branch responsible for U.S. Government systems. These concepts are described more fully elsewhere [11, 13]. The program functions, one for each module access program, give the mapping from the state before the program is executed to the state after the program terminates. The abstraction functions are used to define the "meaning" of the data structure; they give the mapping between the data states and abstract values visible to the users of the module. It is well-known that these functions provide sufficient information for a formal review of correctness of the design before the programs are implemented.

Programs that cannot be described on a single page must be presented in a hierarchical way; each page must present a small program, calling other programs whose functions are specified on that page. This type of presentation allows the algorithm to be understood and verified one page at a time.

If the module embodies a physical model (i.e., a set of equations that allows us to compute nonobservables from observables), the model must be described and its limitations documented.

If the module performs numerical calculations in which accuracy will be a concern, numerical analysis justifying the design must be included.

If the module is hardware-dependent, the documentation must include either a description of the hardware or a reference to such a description.

If the module is responsible for certain parts of the functional specification, a cross reference must be provided.

The reviewers of each internal module design document will include experienced software engineers and other specialists. For example, if a physical model is involved, a physicist or engineer with expertise in that area must be included as a reviewer. If the information is presented in a notation that is independent of the programming language, none of the reviewers needs to be an expert in the programming language involved. Numerical analysts will be needed for some modules, device specialists for others.

What documentation is required to review the code?

While it is important that the algorithms and data structures be appropriate to the task, this will be of little help if the actual code is not faithful to the abstract design. Because of the previous reviews, those who review the code do not need to examine the global design of the system. Instead, they examine the correspondence between the algorithms and the actual code. These reviewers must be experienced users of the hardware and compilers involved; of course, they must also understand the notation used to specify the algorithms.

What documentation is required for the Test Plan Review?

Although these reviews, if carried out rigorously, constitute a mathematical verification of the code, testing is still required. Sound testing requires that a test plan (a document describing the way test cases will be selected) be developed and approved in advance. In addition to the usual engineering practice of normal case and limiting case checks, it is important that the reliability of safety-critical systems be estimated by statistical methods. Reliability estimation requires statistically valid random testing; careful thought must be given to the distribution from which the test cases will be drawn. It is important for the distribution of inputs to be typical of situations in which the correct functioning of the system is critical. A more detailed discussion of statistical testing can be found in the upcoming section, Reliability Assessment for Safety-Critical Software.

The test plan should be described in a document that is not available to the designers. It should be reviewed by specialists in software testing, and specialists in the application area, who compare it with the requirements specification to make certain the test coverage is adequate.

Reviewing the relationship between these documents

The hierarchical process described is designed to allow reviews to be conducted in an orderly way, focusing on one issue at a time. To make this "separation of concerns" work, it is important that the required relationships between the documents be verified.

a. The module guide must show clearly that each of the mathematical functions described in the requirements specification is the responsibility of a specific module. There must be no ambiguity about the responsibilities of the various modules. The module specifications must be consistent with the module guide and the requirements specification.
b. Each module design document should include argumentation showing that the internal design satisfies the module specification. If the module specification is mathematical [18], mathematical verification of the design correctness is possible [11].
c. The module design document, which describes the algorithms, must be clearly mapped onto the code. The algorithms may be described in an abstract notation or via hierarchically structured diagrams.
d. The test plan must show how the tests are derived and how they cover the requirements. The test plan must include black box module tests as well as black box system tests.

Why is configuration management essential for rigorous reviews?

Because of the complexity of software, and the amount of detail that must be taken into consideration, there is always a tremendous amount of documentation. Some of the most troublesome software errors occur when documents are allowed to get out-of-date while their authors work with pencil notes on their own copies.

For the highly structured review process outlined earlier to succeed, all documents must be kept consistent when changes are made. If a document is changed, it, and all documents related to it, must be reviewed again. A careful review of the software may take weeks or months. Each reviewer must be certain that the documents given to him are consistent and up-to-date. The time and energy of reviewers should not be wasted, comparing different versions of the same document.

A process known in the profession as *configuration management*, supported by a configuration control mechanism, is needed to ensure that every designer and reviewer has the latest version of the documents and is informed of every change in a document that might affect the review.

We should be exploiting computer technology to make sure that programmers, designers, and reviewers do not need to retain paper copies of the documents at all. Instead, they use online documentation. If a change must be made, all who have used the affected document should be notified of the change by the computer system. When a change is being considered, but is not yet approved, users of the document should receive a warning. The online versions must be kept under strict control so they cannot be changed without authorization. Every page must contain a version identifier that makes it easier for a reviewer to verify that the documents he has used represent a consistent snapshot.

MODULAR STRUCTURE

Modern software engineering standards call for software to be organized in accordance with a principle

known variously as "Information Hiding," "Object-Oriented Programming," "Separation of Concerns," "Encapsulation," "Data Abstraction," etc. This principle is designed to increase the cohesion of the modules while reducing the "coupling" between modules. Several new textbooks, well-known programming languages such as ADA, practical languages such as MESA, PROTEL, and MODULA, are designed to support such an organization.

Any large program must be organized into programmer work assignments known as modules. In information-hiding designs, each module hides a secret, a fact, or closely related set of facts, about the design that does not need to be known by the writers and reviewers of other modules. Each work assignment becomes much simpler than in an old-fashioned design because it can be completed and understood without knowing much about the other modules. When changes are needed, they do not ripple through an unpredictable number of other modules, as they frequently do in more conventional software designs.

A number of practical systems illustrate the benefits of information hiding even when the designers did not use that abstract principle but depended on their own intuition. For example, the widely used UNIX operating system gains much of its flexibility from hiding the difference between files and devices.

The thought of hiding information from others often strikes engineers as unnatural and wrong. In engineering projects, careful scrutiny by others working on the project is considered an important part of quality control. However, information hiding occurs naturally in large multidisciplinary projects. An electrical engineer may use a transformer without understanding its molecular structure or knowing the size of the bolts that fasten it to a chassis. The circuit designer works with a specification that specifies such abstractions as voltage ratio, hysteresis curve, and linearity. Designers of large mechanical structures work with abstract descriptions of the girders and other components, not with the detailed molecular structures that are the concern of materials engineers. Large engineering projects would be impossible if every engineer on the project had to be familiar with all the details of every component of the product.

Large software projects have the complexity of huge multidisciplinary projects, but there is only one discipline involved. Consequently, information hiding does not occur naturally and must be introduced as an engineering discipline. Software engineers should be trained to provide and use abstract mathematical specifications of components just as other engineers do.

The criterion of information hiding does not determine the software structure. Software engineers try to minimize the information that one programmer must have about another's work. They also try to minimize the expected cost of a system over the period of its use. Both information and expected cost are probabilistic measures. For maximum benefit, one should hide those details most likely to change but does not need to hide facts that are fundamental and unlikely to change. Further, decisions likely to be changed and reviewed together should be hidden in the same module. This implies that to apply the principle, one must make assumptions about the likelihood of various types of changes. If two designers apply the information-hiding principle, but make different assumptions about the likelihood of changes, they will come up with different structures.

RELIABILITY ASSESSMENT FOR SAFETY-CRITICAL SOFTWARE

Should we discuss the reliability of software at all?

Manufacturers, users, and regulatory agencies are often concerned about the reliability of systems that include software. Over many decades, reliability engineers have developed sophisticated methods of estimating the reliability of hardware systems based upon estimates of the reliability of their components. Software is often viewed as one of those components and an estimate of the reliability of that component is deemed essential to estimating the reliability of the overall system.

Reliability engineers are often misled by their experience with hardware. They are usually concerned with the reliability of devices that work correctly when new, but wear out and fail as they age. In other cases, they are concerned with mass-produced components where manufacturing techniques introduce defects that affect only a small fraction of the devices. Neither of these situations applies to software. Software does not wear out, and the errors introduced when software is copied have not been found to be significant.

As a result of these differences, it is not uncommon to see reliability assessments for large systems based on an estimated software reliability of 1.0. Reliability engineers argue that the correctness of a software product is not a probabilistic phenomenon. The software is either correct (reliability 1.0) or incorrect (reliability 0). If they assume a reliability of 0, they cannot get a useful reliability estimate for the system containing the software. Consequently, they assume correctness. Many consider it nonsense to talk about "reliability of software."

Nonetheless, our practical experience is that software appears to exhibit stochastic properties. It is quite useful to associate reliability figures such as MTBF (Mean Time Between Failures) with an operating system or other software product. Some software experts attribute the apparently random behavior to our ignorance. They believe that all software failures would be predictable if we fully understood the software, but our failure to understand our own creations justifies the treatment of software failures as random. However, we know that if we studied the software long enough, we could obtain a complete description of its response to inputs. Even then, it would be useful to talk about the MTBF of the

product. Hence, ignorance should not satisfy us as a philosophical justification.

When a program first fails to function properly, it is because of an input sequence that had not occurred before. The reason that software appears to exhibit random behavior, and the reason that it is useful to talk about the MTBF of software, is because the input sequences are unpredictable. When we talk about the failure rate of a software product, we are predicting the probability of encountering an input sequence that will cause the product to fail.

Strictly speaking, we should not consider software as a component in systems at all. The software is simply the initial data in the computer and it is the initialized computer that is the component in question. However, in practice, the reliability of the hardware is high and failures caused by software errors dominate those caused by hardware problems.

What should we be measuring?

What we intuitively call "software reliability" is the probability of not encountering a sequence of inputs that leads to failure. If we could accurately characterize the sequences that lead to failure we would simply measure the distribution of input histories directly. Because of our ignorance of the actual properties of the software, we must use the software itself to measure the frequency with which failure-inducing sequences occur as inputs.

In safety-critical applications, particularly those for which a failure would be considered catastrophic, we may wish to take the position that design errors that would lead to failure are always unacceptable. In other technologies we would not put a system with a known design error in service. The complexity of software, and its consequent poor track record, means we seldom have confidence that software is free of serious design errors. Under those circumstances, we may wish to evaluate the probability that serious errors have been missed by our tests. This gives rise to our second probabilistic measure of software quality, *trustworthiness.*

In the sequel we shall refer to the probability that an input will not cause a failure as the reliability of the software. We shall refer to the probability that no serious design error remains after the software passes a set of randomly chosen tests as the trustworthiness of the software. We will discuss how to obtain estimates of both of these quantities.

Some discussions about software systems use the terms *availability* and *reliability* as if they were interchangeable. Availability usually refers to the fraction of time that the system is running and assumed to be ready to function. Availability can depend strongly on the time it takes to return a system to service once it has failed. If a system is truly safety-critical (e.g., a shutdown system in a nuclear power station), we would not depend on it during the time it was unavailable. The nuclear reactor would be taken out of service while its shutdown system was being repaired. Consequently, reliability and availability can be quite different.

For systems that function correctly only in rare emergencies, we wish to measure the reliability in those situations where the system must take corrective action, and not include data from situations in which the system is not needed. The input sequence distributions used in reliability assessment should be those that one would encounter in emergency situations, and not those that characterize normal operation.

Much of the literature on software reliability is concerned with estimation and prediction of error-rates, the number of errors per line of code. For safety purposes, such rates are both meaningless and unimportant. Error counts are meaningless because we cannot find an objective way to count errors. We can count the number of lines in the code that are changed to eliminate a problem, but there usually are many ways to alleviate that problem. If each approach to repairing the problem involves a different number of lines (which is usually the case), the number of errors in the code is a subjective, often arbitrary, judgment. Error counts are unimportant because a program with a high error count is not necessarily less reliable than one with a low error count. In other words, even if we could count the number of errors, reliability is not a function of the error count. If asked to evaluate a safety-critical software product, there is no point in attempting to estimate or predict the number of errors remaining in a program.

Other portions of the literature are concerned with reliability growth models. These attempt to predict the reliability of the next (corrected) version on the basis of reliability data collected from previous versions. Most assume the failure rate is reduced whenever an error is corrected. They also assume the reductions in failure rates resulting from each correction are predictable. These assumptions are not justified by either theoretical or empirical studies of programs. Reliability growth models may be useful for management and scheduling purposes, but for safety-critical applications one must treat each modification of the program as a new program. Because even small changes can have major effects, we should consider data obtained from previous versions of the program to be irrelevant.

We cannot predict a software failure rate from failure rates for individual lines or subprograms.

The essence of system-reliability studies is the computation of the reliability of a large system when given the reliability of the parts. It is tempting to try to do the same thing for software, but the temptation should be resisted. The lines or statements of a program are not analogous to the components of a hardware system. The components of a hardware system function independently and simultaneously. The lines of a computer program function sequentially and the effect of one execution depends on the state that results from the earlier executions. One failure at one part of a program may lead to many problems elsewhere in the code.

When evaluating the reliability of a safety-critical software product, the only sound approach is to treat the whole computer, hardware and software, as a black box.

The finite state machine model of programs

The following discussion is based on the simplest and oldest model of digital computing. Used for more than 50 years, this model recognizes that every digital computer has a finite number of states and there are only a finite number of possible input and output signals at any moment in time. Each machine is described by two functions: *next-state*, and *output*. Both have a domain consisting of (state, input) pairs. The range of the next-state function is the set of states. The range of the output function is a set of symbols known as the output alphabet. These functions describe the behavior of a machine that starts in a specified initial state and periodically selects new states and outputs in accordance with the functions.

In this model, the software can be viewed as part of the initial data. It determines the initial state of the programmed machine. Von Neumann introduced a machine architecture in which program and data could be intermixed. Practicing programmers know they can always replace code with data or vice versa. It does not make sense to deal with the program and data as if they were different.

In effect, loading a program in the machine selects a terminal submachine consisting of all states that can be reached from the initial state. The software can be viewed as a finite state machine described by two very large tables. This model of software allows us to define what we mean by the number of faults in the software; it is the number of entries in the table that specify behavior that would be considered unacceptable. This fault count has no simple relation to the number of errors made by the programmer or the number of statements that must be corrected to remove the faults. It serves only to help us to determine the number of tests that we need to perform.

Use of hypothesis testing

In most safety-critical applications we do not need to know the actual probability of failure; we need to confirm the failure probability is very likely to be below a specified upper bound. We propose to run random tests on the software, checking the result of each test. Since we are concerned with safety-critical software, if a test fails (i.e., reveals an error in the software), we will change the software in a way that we believe will correct the error. We will again begin random testing. We will continue such tests until we have sufficient data to convince us that the probability of a failure is acceptably low. Because we can execute only a very small fraction of the conceivable tests, we can never be sure that the probability of failure is low enough. We can, however, calculate the probability that a product with unacceptable reliability would have passed the test that we have carried out.

TABLE I. Probability That a System With Failure Probability of .001 Will Pass N Successive Tests

h=1000.

N	M= (1 - 1/h)^N
500	0.60638
600	0.54865
700	0.49641
800	0.44915
900	0.40639
1000	0.3670
1500	0.22296
2000	0.13520
2500	0.08198
3000	0.04971
3500	0.03014
4000	0.01828
4500	0.01108
4700	0.00907
5000	0.00672

Let us assume the probability of a failure in a test of a program is $1/h$ (i.e., the reliability is $1 - 1/h$). Assuming that N randomly selected tests (chosen, with replacement, from a distribution that corresponds to the actual usage of the program) are performed, the probability there will be no failure encountered during the testing is

$$(1 - 1/h)^N = M. \quad (1)$$

In other words, if we want the failure probability to be less than $1/h$, and we have run N tests without failure, the probability that an unacceptable product would pass our test is no higher than M. We must continue testing, without failure, until N is large enough to make M acceptably low. We could then make statements like, "the probability that a product with reliability worse than .999 would pass this test is less than one in a hundred." Table I provides some sample values of M for $h = 1000$ and various values of N.

Table I shows that, if our design target was to have the probability of failure be less than 1 in 1000, performing between 4500 and 5000 tests (randomly chosen from the appropriate test case distribution) without failure would mean that the probability of an unacceptable product passing the test was less than 1 in a hundred.

Because the probability of failure in practice is a function of the distribution of cases encountered in practice, the validity of this approach depends on the distribution of cases in the tests being typical of the distribution of cases encountered in practice.

We can consider using the same approach to obtain a measure of the trustworthiness of a program. Let the

total number of cases from which we select tests be C. Assume we consider it unacceptable if F of those cases results in faulty behavior; (F might be 1). By substituting F/C for $1/h$ we obtain

$$(1 - F/C)^N = M. \quad (2)$$

We now assume that we have carried out N randomly selected tests without finding an error. If, during that testing, we had found an error, we would have corrected the problem and started again. We can estimate the value of C, and must determine whether to use $F = 1$ or some higher value. We might pick a higher number if we thought it unlikely that there would be only 1 faulty (state, input) pair. In most computer programs, a programming error would result in many faulty pairs, and calculations using $F = 1$ are unnecessarily pessimistic. After choosing F, we can determine M as above. (F, M) pairs provide a measure of trustworthiness. Note that systems considered trustworthy would have relatively low values of M and F.

As a result of such tests we could make statements like, "The probability that a program with more than five unacceptable cases would pass this test is one in a hundred." Since we are not concerned with the frequency of failure of those cases in practice, the tests should be chosen from a distribution in which all state input combinations are equally likely. Because C is almost always large and F relatively small, it is not practical to evaluate trustworthiness by means of testing. Trustworthiness, in the sense that we have defined it here, must be obtained by means of formal, rigorous inspections.

It is common to try to achieve high reliability by using two or more programs in an arrangement that will be safe if one of their specified subsets fails. For example, one could have two safety systems and make sure that each one could alone take the necessary actions in an emergency. If the system failures are statistically independent, the probability of the joint system failing is the product of the probability of individual failures. Unfortunately, repeated experiments have shown that, even when the programs for the two systems are developed independently, the failures are correlated [6, 7, 8]. As a result, we should evaluate the probability of joint failure experimentally.

The hypothesis testing approach can be applied to the evaluation of the probability of joint failures of two (or more) systems. Both systems must be subjected to the same set of test conditions. Joint failures can be detected. However, because the permitted probability of failures for joint systems is much lower than for single systems, many more tests will be needed. Table II shows some typical values.

In this table, we have been quite vague about the nature of a single test and have focused on how many tests are needed. Next we will discuss what constitutes a test and how to select one or more tests.

Three classes of programs

The simplest class of programs to test comprises those that terminate after each use and retain no data from one run to the next. These memoryless batch programs are provided with data, executed, and return an answer that is independent of any data provided in earlier executions.

A second class consists of batch programs that retain data from one run to the next. The behavior of such programs on the nth run can depend on data supplied in any previous run.

A third class contains programs that appear to run continuously. Often these real-time programs are intended to emulate or replace analogue equipment. They consist of one or more processes; some of those processes run periodically, others run sporadically in response to external events. One cannot identify discrete runs, and the behavior at any time may depend on events arbitrarily far in the past.

Reliability estimates for memoryless batch programs: For memoryless batch programs a test consists of a single run using a randomly selected set of input data. If we are concerned with a system required to take action in rare circumstances, and one in which action in other circumstances is inconvenient rather than unsafe, the population of possible test cases should be restricted to those in which the system should take action. It is essential that one know the reliability under those circumstances. Of course, additional tests can be conducted, using other data, to determine the probability of action being taken when no action is required.

Reliability estimates for batch programs with memory: When a batch program has memory, a test consists of a single run. However, a test case is selected by choosing both input data and an internal state. For reliability estimates, the distribution of internal states must match that encountered in practice. It is often more difficult to determine the appropriate distribution of internal states than to find the distribution of inputs. Determining the distribution of internal states requires an understanding of, and experience with, the program.

TABLE II. Probability That a System With Failure Probability of .000001 Will Pass N Successive Tests

h=1000000.		h = 1000000.	
N	M= (1 - 1/h)^N	N	M = (1 - 1/h)^N
1000000.	0.36788	4000000.	0.01832
2000000.	0.13534	4100000.	0.01657
3000000.	0.04979	4200000.	0.01500
4000000.	0.01832	4300000.	0.01357
5000000.	0.00674	4400000.	0.01228
6000000.	0.00248	4500000.	0.01111
7000000.	0.00091	4600000.	0.01005
8000000.	0.00034	4700000.	0.00910
9000000.	0.00012	4800000.	0.00823
10000000.	0.00005	4900000.	0.00745

An alternative to selecting internal states for the test would be to have each test consist of a sequence of executions. The system must be reinitialized before each new sequence. Again, the distribution of these cases must match that found in practice if the reliability estimates are to be meaningful. In addition, it is difficult to determine the length of those sequences. The sequences must be longer than the longest sequence that would occur in actual use. If the sequences are not long enough, the distribution of internal states that occur during the test may be badly skewed. In effect, this means that in actual use, the system must be reinitialized frequently so that an upper bound can be placed on the length of each test.

Reliability estimates for real-time systems: In real-time systems, the concept of a batch run does not apply. Because the real-time system is intended to simulate or replace an analogue system, the concept of an input sequence must be replaced by a multidimensional trajectory. Each such trajectory gives the input values as continuous functions of time. Each test involves a simulation in which the software can sample the inputs for the length of that trajectory.

The question of the length of the trajectory is critical in determining whether or not statistical testing is practical. In many computer systems there are states that can arise only after long periods of time. Reliability estimates derived from tests involving short trajectories will not be valid for systems that have been operating for longer periods. On the other hand, if one selects lengthy trajectories, the testing time required is likely to be impractical.

Statistical testing can be made practical if the system design is such that one can limit the length of the trajectories without invalidating the tests. To do this, one must partition the state. A small amount of the memory is reserved for data that must be retained for arbitrary amounts of time. The remaining data are reinitialized periodically. The length of the period becomes the length of the test trajectory. Testing can then proceed as if the program were a batch program with (memory-state, trajectory) pairs replacing input sequences.

If the long-term memory has a small number of states, it is best to perform statistically significant tests for each of those states. If that is impractical, one must select the states randomly in accordance with a predicted distribution. In many applications, the long-term memory corresponds to operating modes and a valid distribution can be determined.

Picking test cases for safety-critical real-time systems

Particular attention must be paid to trajectory selection if the system is required to act only in rare circumstances. Since the reliability is a function of the input distribution, the trajectories must be selected to provide accurate estimates under the conditions where performance matters. In other words, the population from which trajectories are drawn must include only trajectories in which the system must take action. Similarly, the states of the long-term memory should be restricted to those in which the system will be critical to safety.

Determining the population of trajectories from which the tests are selected can be the most difficult part of the process. It is important to use one's knowledge of the physical situation to define a set of trajectories that can occur. Tests on impossible trajectories are not likely to lead to accurate reliability estimates. However, there is always the danger that the model used to determine these trajectories overlooks the same situation overlooked by the programmer who introduced a serious bug. It is important that any model used to eliminate *impossible* trajectories be developed independently of the program. Most safety experts would feel more comfortable if, in addition to the tests using trajectories considered possible, some statistical tests were conducted with *crazy* trajectories.

CONCLUSIONS

There is no inherent reason that software cannot be used in certain safety-critical applications, but extreme discipline in design, documentation, testing, and review is needed. It is essential that the operating conditions and requirements be well understood, and fully documented. If these conditions are not met, adequate review and testing are impossible.

The system must be structured in accordance with information hiding to make it easier to understand, review, and repair. The documentation must be complete and precise, making use of mathematical notation rather than natural language. Each stage of the design must be reviewed by independent reviewers with the specialized knowledge needed at that stage. Mathematical verification techniques must be used to make the review systematic and rigorous.

An independent agency must perform statistically valid random testing to provide estimates of the reliability of the system in critical situations. Deep knowledge and experience with the application area will be needed to determine the distribution from which the test cases should be drawn.

The vast literature on random testing is, for the most part, not relevant for safety evaluations. Because we are not interested in estimating the error rates or conducting reliability growth studies, a very simple model suffices. Hypothesis testing will allow us to evaluate the probability that the system meets our requirements. Testing to estimate reliability is only practical if a real-time system has limited long-term memory.

Testing to estimate trustworthiness is rarely practical because the number of tests required is usually quite large. Trustworthiness must be assured by the use of rigorous mathematical techniques in the review process.

The safety and trustworthiness of the system will rest on a tripod made up of testing, mathematical review, and certification of personnel and process. In this article, we have focused on two of those legs, testing and review based on mathematical documentation. The

third leg will be the most difficult to implement. While there are authorities that certify professional engineers in other areas, there is no corresponding authority in software engineering. We have found that both classical engineers and computer science graduates are ill-prepared for this type of work. In the long term, those who are concerned about the use of software in safety-critical applications will have to develop appropriate educational programs [15].

Acknowledgments. Conversations with many people have helped to develop these observations. Among them are William Howden, Harlan Mills, Jim Kendall, Nancy Leveson, B. Natvik, and Kurt Asmis. In addition, we are thankful to the anonymous *Communications* referees and the editor for their constructive suggestions.

REFERENCES

1. Britton, K., and Parnas, D. A-7E software module guide. NRL Memo. Rep. 4702, December 1981.
2. Clements, P., Faulk, S., and Parnas, D. Interface specifications for the SCR (A-7E) application data types module. NRL Rep. 8734, August 23, 1983.
3. Currit, P.A., Dyer, M., and Mills, H.D. Certifying the reliability of software. *IEEE Trans. Softw. Eng. SE-12*, 1 (Jan. 1986).
4. Heninger, K. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Trans. Softw. Eng. SE-6*, (Jan. 1980), 2–13.
5. Hester, S.D., Parnas, D.L., and Utter, D.F. Using documentation as a software design medium. *Bell Syst. Tech. J. 60*, 8 (Oct. 1981), 1941–1977.
6. Knight, J.C., and Leveson, N.G. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. Softw. Eng. SE-12*, 1 (Jan. 1986), 96–109.
7. Knight, J.C., and Leveson, N.G. An empirical study of failure probabilities in multi-version software. Rep.
8. Leveson, N. Software safety: Why, what and how. *ACM Comp. Surveys 18*, 2 (June 1986), 125–163.
9. Mills, H.D. Engineering discipline for software procurement. COMPASS '87—Computer Assurance, June 29–July 3, 1987. Georgetown University, Washington, D.C.
10. Mills, H.D. The new math of computer programming. *Commun. ACM 18*, 1 (Jan. 1975), 43–48.
11. Mills, H.D., Basili, V.R., Gannon, J.D., and Hamlet, R.G. *Principles of Computer Programming—A Mathematical Approach*. Allyn and Bacon, Inc., 1987.
12. Mills, H.D., and Dyer, M. A formal approach to software error removal. *J. Syst. Softw.* (1987).
13. Mills, H.D., Linger, R.C., and Witt, B.I. *Structured Programming: Theory and Practice*. Addison-Wesley, Reading, Mass., 1979.
14. Parker, A., Heninger, K., Parnas, D., and Shore, J. Abstract interface specifications for the A-7E device interface module. NRL Memo. Rep. 4385, November 20, 1980.
15. Parnas, D.L. Education for computing professionals. *IEEE Comp. 23* (Jan. 1990), 17–22.
16. Parnas, D.L., and Clements, P.C. A rational design process: How and why to fake it. *IEEE Trans. Softw. Eng. SE-12*, 2 (Feb. 1986), 251–257.
17. Parnas, D.L., Heninger, K., Kallander, J., and Shore, J. Software requirements for the A-7E aircraft. NRL Rep. 3876, November 1978.
18. Parnas, D.L., and Wang, Y. The Trace assertion method of module interface specification. Tech. Rep. 89-261, Queen's University, TRI (Telecommunications Research Institute of Ontario). October 1989.
19. Parnas, D.L., and Weiss, D.M. Active design reviews: Principles and Practices. In *Proceedings of the 8th International Conference on Softw Engineering* (London, August 1985).

ABOUT THE AUTHORS:

DAVID L. PARNAS is professor of Computing and Information Science at Queen's University in Kingston, Ontario. His work interests involve most aspects of computer system engineering. His special interests include precise abstract specifications, real-time systems, safety-critical software, program semantics, language design, software structure, process structure, and process synchronization.

A. JOHN VAN SCHOUWEN, currently completing his master's thesis at Queen's University, is a research associate at the Telecommunications Research Institute of Ontario. His research interests include formal and precise software documentation.

Authors' Present Address: Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6.

SHU PO KWAN is a specialist in nuclear reaction and nuclear structure. He has also done research work in computer simulation and modelling. Author's Present Address: 1118 Avenue Rd., Toronto, Ontario, Canada M5N 2E6.